










# EPMC Gets Knowledge in Multi-agent Systems



Chen Fu<sup>1,2</sup>(✉) , Ernst Moritz Hahn<sup>3</sup> ,  
Yong Li<sup>1</sup> , Sven Schewe<sup>4</sup> , Meng Sun<sup>5</sup> ,  
Andrea Turrini<sup>1,6</sup> , and Lijun Zhang<sup>1,2,6</sup> 

<sup>1</sup> State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China  
fchen@ios.ac.cn

<sup>2</sup> University of Chinese Academy of Sciences, Beijing, China

<sup>3</sup> University of Twente, Enschede, The Netherlands

<sup>4</sup> University of Liverpool, Liverpool, UK

<sup>5</sup> LMAM and Department of Information Science, School of Mathematical Sciences, Peking University, Beijing, China

<sup>6</sup> Institute of Intelligent Software, Guangzhou, China


**Abstract.** In this paper, we present EPMC, an extendible probabilistic model checker. EPMC has a small kernel, and is designed modularly. It supports discrete probabilistic models such as Markov chains and Markov decision processes. Like PRISM, it supports properties specified in PCTL\*. Two central advantages of EPMC are its modularity and extendibility. We demonstrate these features by extending EPMC to EPMC-PETL, a model checker for probabilistic epistemic properties on multi-agent systems. EPMC-PETL takes advantage of EPMC to provide two model checking algorithms for multi-agent systems with respect to probabilistic epistemic logic: an exact algorithm based on SMT techniques and an approximated one based on UCT. Multi-agent systems and epistemic properties are given in an extension of the modelling language of PRISM, making it easy to model this kind of scenarios.

## 1 Introduction

In this paper, we present a new model checker called EPMC, an acronym for *Extendible Probabilistic Model Checker*. Two main characteristics of EPMC are its high modularity and its full extendibility. It achieves its flexibility by an infrastructure that consists of a minimal core part and multiple plugins that

---

This work was supported in part by the Guangdong Science and Technology Department (Grant No. 2018B010107004) and by the National Natural Science Foundation of China (Grants Nos. 62102407, 62172019).

 This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements 864075 (CAESAR), and 956123 (FOCETA).

provide model checking functionalities. We believe that it is very convenient to develop a new model checker based on the core parts of EPMC. While the model checker historically starts from probabilistic models, it will be easy to extend it to incorporate other model types.

The baseline includes model checking functionality for probabilistic systems. Probabilistic systems play an important role in reasoning about randomised network protocols, and biological and concurrent systems. They also find applications in analysing security protocols. Markov decision processes are among the most important semantic models. As a result, several model checkers that support MDP analysis have been developed, including the state-of-the-art probabilistic model checker PRISM [35], STORM [15], MRMC [30], LIQUOR [12], MOCHIBA [50], and ISCASMC [22]. These model checkers differ in the model and property types they support. For instance, MRMC and STORM handle branching time properties specified in PCTL [25], whereas LIQUOR, ISCASMC and MOCHIBA are specialised in analysing linear time properties (PLTL) [5]. PRISM can handle both.

The first baseline of EPMC includes support for PCTL, PLTL, and their extension to PCTL\*. In addition, it can also be used to analyse Markov games. To demonstrate the main features of EPMC, we extend it to the model checker EPMC-PETL. EPMC-PETL is designed for the verification of probabilistic multi-agent systems against PETL (probabilistic epistemic temporal logic) properties under uniform schedulers. Multi-agent systems have found many applications and verification techniques have also been proposed over the past decades. Although there are model checkers for multi-agent systems, as we will see in related works (Sect. 4), they can only handle restricted classes of the model we are interested in, such as a non-probabilistic setting or, where they can handle probability, they do not support epistemic accessibility relations. The algorithmic design, implementation, and validation based on an existing model checker for probabilistic multi-agent systems against properties specified in PETL under uniform schedulers is not available.

Exploiting the minimal kernel and multiple plugins of EPMC, we can conveniently implement the algorithms specific for the epistemic fragment of PETL while reusing the core parts of EPMC for the management of the remaining fragment, part of PCTL. In particular, the modularity of EPMC makes the development of new functionalities rather independent from the existing ones, without having to change existing code. This speeds up the implementation and simplifies the debugging of the code, by isolating the different components responsible for the different verification steps.

Summarising, the main features of EPMC include extendibility, modularity, and the support of games and strategy synthesis. Beyond introducing EPMC, we also present, with its extension to EPMC-PETL, *the first tool* that supports PETL model checking for probabilistic nondeterministic multi-agent systems.

*Organisation of the Paper.* Section 2 introduces the architecture of our tool. In particular, we demonstrate how to develop the PETL model checker EPMC-PETL. Experimental results are presented in Sect. 3. Sect. 4 discusses related works, and Sect. 5 concludes the paper.

## 2 Architecture

We show the architecture of EPMC and how to build EPMC-PETL on top of it. EPMC contains two main components: a) EPMC core; b) various plugins. Details of these components and the interface are provided below.

The larger part of EPMC is developed in Java. It uses JNA [3] to access libraries written in C/C++ to improve the performance of some computation or to provide access to legacy code. Instances of such libraries are the BDD libraries (like CUDD [51]) used to store symbolically the models or the C implementation of different versions of value iteration algorithms. The compilation of EPMC is managed by the software project management and comprehension tool Maven [2]. Maven takes care of caching and retrieving all building dependencies such as Ant [1] and JavaCC [4], used for the parsers. This allows for porting EPMC to multiple platforms and architectures.

### 2.1 EPMC Core

EPMC consists of a minimal kernel and multiple plugins that provide the functionalities needed for model checking. This kernel is rather small. It is only responsible for the bootstrap phase, where the plugins are loaded, and for starting the model checking procedure. It first initialises the data structures needed to load the plugins and then loads and initialises each plugin according to the order, in which they are specified. Finally, it starts the model checking procedure by parsing the given models and properties and calling the appropriate solvers.

In order to maximise modularity, the kernel has no information about the existing plugins until they are loaded and initialised; it is the duty of each plugin to register itself in EPMC. In order to be recognised as a valid EPMC plugin, it has to

- declare its name and that it is an EPMC plugin in its `MANIFEST.MF` file;
- list the plugins it depends on; and
- implement all interfaces defined by the plugin manager from the kernel part.

Once the plugin meets these requirements, it can be used in EPMC to provide the expected functionalities. The plugin can be inserted into EPMC in two ways: either its `jar` file is placed in the `embeddedplugins` directory contained in the EPMC `jar` file and its name is listed in the `embeddedplugins.txt` file; or it is specified at command line by means of the option `plugin` as a `jar` file or as a directory containing the class files. During the kernel's bootstrap phase, the plugins listed in `embeddedplugins.txt` are loaded first, following the order in which they appear in the file. Then the plugins specified by the option `plugin` are loaded according to their order.

When loading a plugin, a set of specific methods defined by the plugin interface are called. In these methods, the plugin can register itself with respect to its functionalities. A plugin can, for example, add new command line options, new commands, or new data types; or it can declare to support specific operations,

such as model checking a specific logic operator. The registration performed by a plugin can be altered by the plugins loaded later. A plugin loaded later has therefore a higher priority than a plugin loaded earlier. In particular, one can last load a simple plugin that removes or modifies some of the options provided earlier in order to create a version of EPMC specialised for specific tasks within a specific setting.

## 2.2 Plugins Available in EPMC

We will now introduce some of the plugins natively supported by EPMC; the different flavours of EPMC can be obtained by choosing and combining multiple plugins together: for instance, by selecting the appropriate set of plugins EPMC becomes a tool for performing PCTL model checking on Markov chains or MDPs, and with a different set of plugins we can obtain a tool for model checking Markov decision processes against PLTL formulas. By combining the two sets of plugins, the resulting EPMC is able to check these models against the whole of PCTL\*. Below we give an overview of the plugins of EPMC.

*Algorithm Group:* This group contains all plugins that provide the classical algorithms that are used for probabilistic model checking, such as graph decomposition into strongly connected components and maximal end components for both symbolic and explicit representations. It currently only includes the plugin **algorithm**, which provides standard algorithms, such as the following ones: FOXGLYNN, which follows the algorithm proposed in [28] for computing Poisson probabilities for CTMCs; TARJAN, which implements the well-known strongly connected component decomposition algorithm by Robert Tarjan [53] for explicit data structures; and BLOEM and CHATTERJEE, which compute strongly connected components using BDDs and are based on the work of Roderick Bloem et al. [6] and Krishnendu Chatterjee et al. [10], respectively.

*Automata Group:* The purpose of this group is to enclose the plugins that encode  $\omega$ -regular automata. It currently includes two plugins, namely the **automata** and the **automaton-determinisation** plugins. **automata** provides a uniform interface for automata such as Büchi and Rabin automata, while **automaton-determinisation** provides the algorithms proposed by Sven Schewe, Thomas Varghese, and Nir Piterman [46, 48, 49] to determinise nondeterministic Büchi automata to deterministic Rabin and parity automata.

*Command Group:* This group provides three plugins that set the main functionality of EPMC: **command-check** calls the model checker to actually perform the model checking operation; **command-help** prints out the usage messages; and **command-lump** requires as input a probabilistic model and generates as output a new model, which is bisimilar to the original model.

*BDD Group:* The BDD group is dedicated to the symbolic representation of models and properties by means of the Binary Decision Diagrams data structures. The **dd** plugin provides a uniform interface to use a BDD library and therefore does not provide any actual implementation of BDD data structures. Such an implementation is provided by one of the following plugins; each of them implements the dd interface and at least one of them has to be included whenever EPMC is expected to support the symbolic representation of models.

The **dd-buddy** plugin wraps the C library BuDDy [14], which is a small and efficient BDD library. The **dd-cacbdd** plugin gives access to the C++ library CacBDD [44], which implements a dynamic cache management algorithm. The **dd-cudd** plugin provides the C library CUDD [51], which is the most well-known BDD library used in several tools; it is the default BDD library of the PRISM model checker [35,47]. The **dd-cudd-mtbdd** plugin is the companion of **dd-cudd** for the multi-terminal binary decision diagrams (MTBDDs) offered by CUDD. The **dd-jdd** plugin includes the library JDD [54], which is a Java implementation of binary decision diagrams inspired by BuDDy. The **dd-sylvan** and **dd-sylvan-mtbdd** plugins make the library Sylvan [17] available in EPMC; Sylvan is a parallel (multi-core) BDD library written in C.

*Bisimulation Algorithm Group:* This group collects the plugins that compute bisimulation relations on the models: the **lumper-explicit-signature** plugin implements a signature based lumping algorithm for probabilistic systems; and the **lumper-dd** plugin implements a lumping algorithm for probabilistic systems by using MTBDDs.

*Expression Group:* This group hosts the **expression-basic** plugin, which is designed to provide a uniform interface as well as the corresponding data structures to handle formulas from temporal logics like PCTL and PLTL.

*Graph Group:* The single **graph** plugin available in this group provides a uniform interface as well as the data structures to store various models as a graph. The model can be a Markov chain, a Markov decision process, an automaton, or any model that can be interpreted as a labelled graph. It also provides the interfaces to access the properties in the nodes or the properties on the edges. For instance, it permits to collect all atomic propositions that hold in a state via evaluating the properties of this state node.

*Graph Solver Group:* Similar to the BDD group, we have the **graphsolver** plugin, which defines a uniform interface for solving the linear programming problems used to compute the reachability probabilities the model checking problems are reduced to. The actual implementation is provided by the **graphsolver-iterative** plugin, which solves the given linear programming problem by value iteration. It supports both JACOBI and GAUSS-SEIDEL iteration methods.

*JANI Format Group:* This group contains all plugins related to the recently proposed JANI model and interaction format [7]. There are currently three plugins:

the **jani-model** plugin provides a parser to transform an input JANI model to a graph or an MTBDD. It is also able to parse the input JANI formula; the **jani-exporter** takes care of exporting models and properties in JANI format.

*PRISM Format Group:* The single **prism-format** plugin available in this group provides a parser to transform a given PRISM model description to an explicit graph or an MTBDD. It also provides a parser for the input formula.

*Property Solver Group:* The plugins contained in this group are responsible for solving the properties analysed during the model checking phase. Similarly to the BDD group, the specific property solvers are all implementations of the common interface provided by the **propertysolver** plugin. There are currently eight implementation plugins representing eight different classes of properties: **propertysolver-coalition** provides a solution to solve a probabilistic parity game against linear temporal properties; **propertysolver-filter** handles the filter operation in the given PRISM formula; **propertysolver-ltl-lazy** implements an efficient method to model check the PCTL\* logic over the probabilistic systems by means of advanced LTL verification techniques; **propertysolver-operator** works with the operators that occur in the given formula; **propertysolver-pctl** implements the PCTL model checking algorithm over probabilistic systems; **propertysolver-propositional** provides a way to identify all states that satisfy the given propositional formula; **propertysolver-reachability** exemplifies how to write a plugin that handles the reachability formula  $\mathbb{P}_{Fa}$  over Markov chains; and **propertysolver-reward** implements a model checking algorithm to handle probabilistic systems with rewards.

*Util Group:* The single plugin **util** available in this group provides basic utilities useful for working with bits, JSON documents, and other native data types in a JAVA-style approach.

*Value Group:* Similar to the expression group, this group hosts the **value-basic** plugin, which is designed to provide a uniform interface to represent all kinds of values and types that may be used in EPMC, as well as the implementation of the standard values and type such as Booleans, integers, and reals.

**Dependencies Between Plugins.** Each plugin may have build-time and runtime dependencies on other plugins. Build-time dependencies can be considered as hard dependencies: they must be satisfied at compilation time as well as during the bootstrap phase; these build-time dependencies are made explicit in the `MANIFEST.MF` file, and the order the plugins are loaded in the bootstrap phase has to respect such build-time dependencies. For instance, the **property-solver-pctl** plugin has a build-time dependency on **property-solver**, since **property-solver-pctl** implements the interfaces defined by **property-solver**.

The graph of build-time dependencies between the groups of plugins is shown in Fig. 1, where an arrow from one group to another means that the former

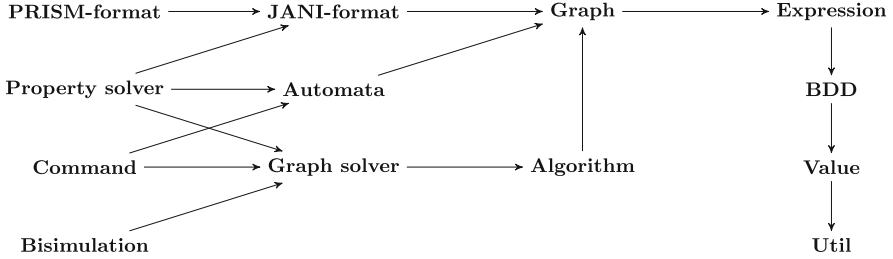


Fig. 1. Build-time dependencies between groups of plugins in EPMC

requires the latter. To simplify the graph, we omitted all arrows that can be inferred by transitivity, such as the one between any group and **util**.

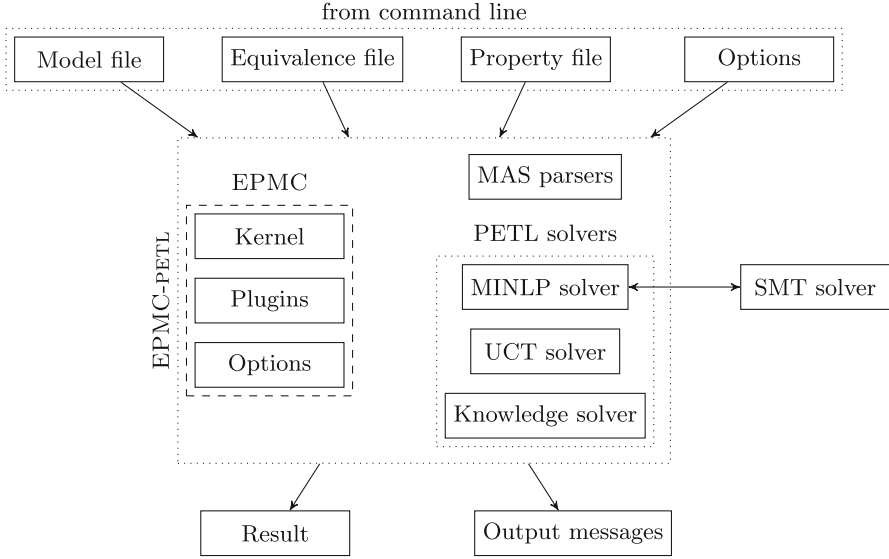
Run-time dependencies can be seen as soft dependencies: their satisfaction depends on the actual steps performed during the model checking phase. For instance, the **property-solver-pctl** plugin has only a run-time dependency on **graphsolver-iterative**, since **graphsolver-iterative** is required during the model checking phase only in cases the property cannot be decided via a simple graph exploration. (This happens for quantitative properties.) This means that **graphsolver-iterative** has to be available at run-time for some properties, while for other properties it may be missing. If EPMC is intended to be used to check only qualitative properties, then **graphsolver-iterative** can safely be omitted, while EPMC needs the **graphsolver-iterative** plugin (or any other plugin implementing **graphsolver**) to analyse quantitative properties.

### 2.3 PETL Model Checker as a Plugin

The structure of EPMC-PETL, largely shared with EPMC given its modular architecture, is illustrated in Fig. 2.

To provide the PETL model checking algorithms for multi-agent systems offered by EPMC-PETL, we have developed the PETL plugins that add the corresponding functionalities, namely: the parser for the multi-agent system model specification and the PETL properties; the data structures to store them; and the algorithms for evaluating the properties against the given model.

In multi-agent systems, the agents have the capacity to perform certain actions, which they choose according to their individual protocols. Given the distributed nature of multi-agent systems, it is typical that the agents have incomplete information about the state of the global system due to the fact that they are only able to observe a limited part of the global state when they have to choose their actions. The incompleteness of information is normally modelled by defining, for each agent  $i$ , an equivalence relation  $\sim_i$  over all global states of the systems, then two global states are considered indistinguishable for a given agent  $i$  if they are related by  $\sim_i$ . Note that two states that are indistinguishable for an agent may be distinguishable for another agent, so there is no constraint on how two states are related by the different relations. Every agent makes its own



**Fig. 2.** Architecture of EPMC-PETL

decisions based only on the limited information it has, namely, the information restricted by its own indistinguishability relation. Decisions of agents are usually formalised by *schedulers*, which are functions that take the history executions as input and decide (output) the next move for each agent. Schedulers that only make use of the limited information each agent is aware of are called *uniform*. Intuitively, a uniform scheduler for the agent  $i$  is expected to make the same choice when given two executions that are equivalent under  $\sim_i$ .

To build the model checker EPMC-PETL, we have to first implement three things in this plugin: the model, the property, and the equivalence relations.

*Model.* We use the PRISM language as input format and the model type should be “mdp”, to represent the fact that the model has both probabilistic and non-deterministic behaviour. Each module in the MDP defines one agent’s behaviour, with the name of the module being the agent’s name. The state space of the overall multi-agent system is constructed following the PRISM approach, i.e., by considering all state variables, whether local to a module or global, and with the usual PRISM restrictions on how transitions can update these variables.

Differently from the standard PRISM language semantics, at each step every agent chooses one action among the enabled transitions, independent of whether other agents have a transition with the same action that is enabled. The actions labelling the transitions are therefore not used for the synchronisation of the modules: they are instead the names of local actions, and each command must be labelled by one action.



The overall result is that the agents do not interact with each other by synchronising on common actions, but by the effects of the individual transitions chosen by the individual agents.

*Property.* To specify the properties of probabilistic multi-agent systems, in particular the temporal dynamics of agents’ knowledge, we adopt the probabilistic epistemic temporal logic (PETL) (cf. [16]), which can be viewed as a combination of epistemic logic [18] and probabilistic computation tree logic (PCTL) [25]. To specify PETL formulas, we extend the PRISM language by adding the epistemic operators  $\mathbf{K}_i$  and  $\mathbf{E}_G$ ,  $\mathbf{C}_G$ , and  $\mathbf{D}_G$  to the set of operators that can occur in a property formula, where  $i$  is (the name of) an agent and  $G$  is a set of agents. Intuitively, the property  $\mathbf{K}_i\varphi$  means that agent  $i$  knows that property  $\varphi$  holds in state  $s$  if  $\varphi$  holds in all states equivalent to  $s$  with respect to  $\sim_i$ ; properties  $\mathbf{E}_G\varphi$ ,  $\mathbf{C}_G\varphi$ , and  $\mathbf{D}_G\varphi$  are similar, but refer to the common/distributed knowledge of the group of agents. These epistemic operators are thus added to the PRISM properties as  $\mathbf{K}$  {agent} and  $\mathbf{E}/\mathbf{C}/\mathbf{D}$  {agent<sub>1</sub>, . . . , agent<sub>n</sub>}, respectively.

*Equivalence Relations.* Equivalence relations are encoded as sets of formulas shown in Fig. 3. Each agent in the model has its own `equiv agent_name . . . equiv end` block and each block contains a set of formulas. The formulas are defined on all state variables that occur in the model definition and are not restricted to those of the corresponding single agent.

Each formula induces one equivalence class, i.e., two states that satisfy the same formula of agent  $j$  are considered to be related by  $\sim_j$ . This means that formulas are required to be pairwise disjoint; if a state does not satisfy any formula, it is not equivalent to any other state, so it belongs to its singleton equivalence class.

*PETL Solvers.* In general, the model checking problem for probabilistic multi-agent systems against PETL properties is undecidable [20], but is decidable when restricted to the class of uniform memoryless schedulers. The decision algorithm for the latter follows the PCTL approach: the PETL property is checked bottom-up, with each operator managed by its corresponding solver. Epistemic operators are part of the state formulas while the temporal operators are managed as in PCTL, except for the class of schedulers considered for computing the Until operator.

The key parts of the PETL plugins are three solvers needed to verify PETL properties: the first one focuses on the knowledge operators, while the other two take care of the PCTL until operator (wrapped inside a probabilistic operator  $\mathbb{P}$ , as in PCTL), which needs to be computed on the class of uniform memoryless schedulers instead of the general class of memoryless schedulers as done in PCTL; these two solvers implement two different algorithms, an exact one based on mixed integer non-linear programming and an approximation based on upper

```

equiv agent1
-- formula1;
-- formula2;
      :
equiv end
      :
equiv agentN
-- formula1;
-- formula2;
      :
equiv end

```

**Fig. 3.** The format of equivalence relations

confidence bounds applied to trees (UCT) [31]. The remaining fragments of PETL, like propositional formulas and the next operator, can be computed as for PCTL. They can therefore be inherited from the existing plugins of EPMC.

*MINLP Solver.* This solver implements the PETL model checking algorithm developed in [20]: it reduces the problem of checking the satisfaction of an until formula to a mixed integer non-linear programming (MINLP) problem, which can then be solved by, e.g., an SMT solver. Here we make use of the SMT solver Z3 [45], which can be replaced by any other SMT solver that supports SMT-lib version 2.5 as input format. The reduction ensures that the resulting scheduler is uniform and memoryless, with a different encoding for  $\mathbb{P}_{\max=?}$  and  $\mathbb{P}_{\min=?}$ .

*UCT Solver.* This solver implements an approximated algorithm relative to the until operator, based on the upper confidence bounds applied to trees (UCT) algorithm [19]. This UCT based solver performs a Monte Carlo sampling of the model, with heuristics guiding the choice between the exploration of new parts of the state space, the analysis of already explored state space, and the action to choose. This solver offers several parameters to the user to tune the heuristics: time limit – how much time the solver should use when exploring the model; depth limit – how many steps the solver should perform in the state space exploration; B value – the bias parameter in the UCT formula between old and new state exploration; and random seed – the random seed used to select unvisited successors (so to be able to reproduce the solver’s execution).

The implementation of this solver makes use of specialised data structures to store the information collected during the UCT sampling; in particular, the data structure organises the information so to ensure that the underlying scheduler is uniform, as required by the PETL decision algorithm. The basic idea is to store the selected actions of each agent, and then exclude the actions making the scheduler non-uniform when executing the next step in the exploration.

*Knowledge Solver.* This solver deals with the knowledge operators, namely  $\mathbf{K}_i$ ,  $\mathbf{E}_G$ ,  $\mathbf{D}_G$ , and  $\mathbf{C}_G$ . Depending on the actual knowledge property  $\mathbf{Z}(\varphi)$ , the solver takes the satisfaction information about the state formula  $\varphi$  already computed (recall that PETL model checking is based on a bottom-up approach similar to PCTL) and returns the set of states that satisfy  $\mathbf{Z}(\varphi)$ , by implementing the semantics of  $\mathbf{Z}(\varphi)$ .

**Online Availability.** EPMC, including its extension EPMC-PETL, is an open source tool. EPMC is freely available at <https://github.com/ISCAS-PMC/ePMC> as a git repository to be forked and modified.

### 3 Empirical Evaluation

We have generated five different flavours of EPMC by loading different modules. One version that supports only PCTL; one that supports PCTL\*; one

**Table 1.** Different variations of EPMC. The runtime is given in seconds, and ‘ns’ and ‘to’ abbreviate ‘not supported’ and ‘time-out’ (set to 100s, as performance was not our concern). The properties used were  $\varphi_1 = \mathbb{P}_{\max=?}[\mathbf{F}\text{num\_crit} > 1]$  (PCTL);  $\varphi_2 = \mathbb{P}_{\min=?}[(\mathbf{GFp1!} = 10 \vee \mathbf{GFp1} = 0 \vee \mathbf{FGp1} = 1) \wedge \mathbf{GFp1!} = 0 \wedge \mathbf{GFp1} = 1]$  (PLTL);  $\varphi_3 = \mathbb{P}_{>=1}[\mathbf{F}$  “premium”] (PCTL);  $\varphi_4 = \mathbb{P}_{=?}[(\mathbf{GFleft\_n} = 16) \vee \bigvee_{i=13}^{16} \mathbf{FGrigh\_n} = i]$  (PLTL);  $\varphi_5 = \langle\langle 1 \rangle\rangle_{>=1}[(!“z1” \mathbf{U} “z2”)]$  (Coalition);  $\varphi_6 = \langle\langle 1 \rangle\rangle_{>=1}[(!“z1” \mathbf{U} “z2”) \wedge \mathbf{F}“z3”]$  (Coalition);  $\varphi_7 = \langle\langle 1 \rangle\rangle_{\min=?}[(!“z1” \mathbf{U} “z2”)]$  (Coalition);  $\varphi_8 = \langle\langle 1 \rangle\rangle_{\max=?}[(!“z1” \mathbf{U} “z2”) \wedge (!“z4” \mathbf{U} “z2”) \wedge \mathbf{F}“z3”]$  (Coalition);  $\varphi_9 = \mathbb{P}_{\max=?}[\mathbf{G}(rw\_x \neq rc\_x \vee rw\_y \neq rc\_y)]$  (PETL); and  $\varphi_{10} = \mathbb{P}_{\max=?}[\mathbf{GE}_{rw,rc}(rw\_x \neq rc\_x \vee rw\_y \neq rc\_y)]$  (PETL).

Experiment		EPMC					PRISM	Rabinizer4	PRISM-games
		PCTL	PCTL*	SMG	PETL	full			
Mutual	$\varphi_1$	1.7	1.8	ns	ns	1.8	0.0	0.0	0.0
Exclusion 4	$\varphi_2$	ns	4.5	ns	ns	4.5	14.4	10.4	13.3
Workstation	$\varphi_3$	1.1	1.0	ns	ns	1.2	0.0	0.0	0.0
Cluster 16	$\varphi_4$	ns	1.8	ns	ns	1.7	to	0.7	to
Robot	$\varphi_5$	ns	ns	2.9	ns	2.9	ns	ns	0.6
10	$\varphi_6$	ns	ns	3.1	ns	3.2	ns	ns	1.9
Robot_shoot	$\varphi_7$	ns	ns	5.2	ns	5.7	ns	ns	0.0
7, 1, 0.3	$\varphi_8$	ns	ns	5.9	ns	5.5	ns	ns	2.0
Reconnaissance	$\varphi_9$	ns	ns	ns	17.1	12.6	ns	ns	ns
2	$\varphi_{10}$	ns	ns	ns	16.4	15.3	ns	ns	ns

for solving probabilistic parity games; one that supports PETL; and a version that supports all of these. As comparison, we considered the following tools PRISM [35], PRISM-games [11], and Rabinizer4 [34].

We have run these tools on a few MDP benchmarks taken from the PRISM website [47], SMG games from [23, 24] and multi-agent systems from [20]; we considered some simple properties for these models. The goal of the comparison, reported in Table 1, is to show the adaptability of EPMC in supporting different logics and to use different modules, not the actual performance.

## 4 Related Work

We have already discussed related probabilistic model checkers in the introduction, all of which do not support PETL model checking. Here we list related tools for analysing multi-agent systems or epistemic logics.

MCMAS [41–43] is an open-source, OBDD-based symbolic model checker for verifying multi-agent systems. MCMAS is restricted to non-probabilistic models. There are some model checkers for multi-agent systems built on top of MCMAS: MCMAS-SDD [36] introduces an SDD-based technique for the formal verification of multi-agent systems; MCMAS-SLK [8] supports the verification of systems against specifications expressed in strategy logic (SL) with knowledge; MCMAS-SL[1G] [9] puts forward an automata-based methodology for verifying and synthesising multi-agent systems against specifications given in

SL[1G], which is the one-goal fragment of strategy logic;  $\text{MCMAS}_{\text{LDLK}}$  [32] can verify properties given in LDLK (Linear Dynamic Logic with Knowledge) for multi-agent systems;  $\text{MCMAS}_{\text{LDL}_fK}$  [33] implements the algorithm for the verification of multi-agent systems against  $\text{LDL}_fK$  specifications, which is LDLK interpreted on finite traces. As for MCMAS, all these model checkers do not consider probabilistic components in their systems and logics.

Probabilistic swarm systems support systems with an unbounded and time-changing number of agents. Based on PRISM [35], Lomuscio and Pirovano have introduced the software package PSV (probabilistic swarm verifier), with several sub-components that support bounded time PSV-BD [38], counter abstraction PSV-CA [39], strategic properties PSV-S [40], and faulty systems. The logics these tools consider are either without epistemic operators, or they allow only a single epistemic operator to occur as the top operator of the formula. While the EPMC extension EPMC-PETL we have discussed only analyses systems with a fixed number of agents, it supports the nesting of epistemic operators as well as their boolean combination.

MCK [21] is an OBDD-based model checker for multi-agent systems that supports temporal-epistemic specifications. It has been extended in [26] to support probabilistic reasoning, but nondeterministic choices are not considered; the work in [27] implements a symbolic BDD-based model checking algorithm for an epistemic strategy logic with observational semantics also based on MCK. Epistemic accessibility relations are studied in this work, but only for a non-probabilistic setting. EPMC-PETL supports the analysis of systems that combine nondeterminism and probabilistic choices, which is missing in these tools.

MCTK [52] is a symbolic model checker for a temporal logic of knowledge. It is developed from NuSMV [13]. Similarly, the authors of [37] propose a methodology for model checking a temporal-epistemic logic by building upon an extension of NuSMV. Verics [29] is a model checker for real-time and multi-agent systems. It implements bounded model checking algorithms for CTL, real-time CTL, and variants of CTL that include epistemic operators. Again, these tools can only work with non-probabilistic multi-agent systems.

## 5 Conclusion

In this paper we have presented EPMC, an extendible probabilistic model checker, and EPMC-PETL, a tool for model checking epistemic properties on multi-agent systems that exhibit both probabilistic and nondeterministic behaviours. Key advantages of EPMC are its high degree of modularity and full extendibility. We have exemplified by the particular extension of EPMC-PETL how this extensibility can be used to easily cover attractive new properties that no other solver has covered before. Of course, besides demonstrating this advantage of EPMC, EPMC-PETL also provides this additional functionality, which is novel and a contribution in itself.

## References

1. Apache Ant™ website. <http://ant.apache.org/>
2. Apache Maven website. <http://maven.apache.org/>
3. Java Native Access (JNA) website. <https://github.com/java-native-access/jna>
4. JavaCC™: The Java Compiler Compiler™ website. <http://javacc.org/>
5. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995). <https://doi.org/10.1007/3-540-60692-0-70>
6. Bloem, R., Gabow, H.N., Somenzi, F.: An algorithm for strongly connected component analysis in  $n \log n$  symbolic steps. *Formal Methods Syst. Des.* **28**(1), 37–56 (2006)
7. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 151–168. Springer, Heidelberg (2017). [https://doi.org/10.1007/978-3-662-54580-5\\_9](https://doi.org/10.1007/978-3-662-54580-5_9)
8. Čermák, P., Lomuscio, A., Mogavero, F., Murano, A.: MCMAS-SLK: a model checker for the verification of strategy logic specifications. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 525–532. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_34](https://doi.org/10.1007/978-3-319-08867-9_34)
9. Cermák, P., Lomuscio, A., Murano, A.: Verifying and synthesising multi-agent systems against one-goal strategy logic specifications. In: AAI, pp. 2038–2044 (2015)
10. Chatterjee, K., Henzinger, M., Joglekar, M., Shah, N.: Symbolic algorithms for qualitative analysis of Markov decision processes with Büchi objectives. *Formal Methods Syst. Des.* **42**(3), 301–327 (2013)
11. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: a model checker for stochastic multi-player games. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 185–191. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-36742-7\\_13](https://doi.org/10.1007/978-3-642-36742-7_13)
12. Ciesinski, F., Baier, C.: Liquor: a tool for qualitative and quantitative linear time analysis of reactive systems. In: QEST, pp. 131–132 (2006)
13. Cimatti, A., et al.: NuSMV 2: an OpenSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45657-0-29>
14. Cohen, H., Whaley, J., Wildt, J., Gorogiannis, N.: BuDDy. <http://sourceforge.net/p/buddy/>
15. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A STORM is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-63390-9\\_31](https://doi.org/10.1007/978-3-319-63390-9_31)
16. Delgado, C., Benevides, M.: Verification of epistemic properties in probabilistic multi-agent systems. In: Braubach, L., van der Hoek, W., Petta, P., Pokahr, A. (eds.) MATES 2009. LNCS (LNAI), vol. 5774, pp. 16–28. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-04143-3\\_3](https://doi.org/10.1007/978-3-642-04143-3_3)
17. van Dijk, T., van de Pol, J.: Sylvan: multi-core decision diagrams. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 677–691. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_60](https://doi.org/10.1007/978-3-662-46681-0_60)
18. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (2004)

19. Fu, C., Turrini, A., Huang, X., Song, L., Feng, Y., Zhang, L.: Model checking for probabilistic multiagent systems under uniform schedulers, submitted for publication, shared by the authors
20. Fu, C., Turrini, A., Huang, X., Song, L., Feng, Y., Zhang, L.: Model checking probabilistic epistemic logic for probabilistic multiagent systems. In: IJCAI, pp. 4757–4763 (2018)
21. Gammie, P., van der Meyden, R.: MCK: model checking the logic of knowledge. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 479–483. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-27813-9\\_41](https://doi.org/10.1007/978-3-540-27813-9_41)
22. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: ISCASMC: a web-based probabilistic model checker. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 312–317. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-06410-9\\_22](https://doi.org/10.1007/978-3-319-06410-9_22)
23. Hahn, E.M., Schewe, S., Turrini, A., Zhang, L.: A simple algorithm for solving qualitative probabilistic parity games. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 291–311. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-41540-6\\_16](https://doi.org/10.1007/978-3-319-41540-6_16)
24. Hahn, E.M., Schewe, S., Turrini, A., Zhang, L.: Synthesising strategy improvement and recursive algorithms for solving 2.5 player parity games. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 266–287. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-52234-0\\_15](https://doi.org/10.1007/978-3-319-52234-0_15)
25. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *FAC* **6**(5), 512–535 (1994)
26. Huang, X., Luo, C., van der Meyden, R.: Symbolic model checking of probabilistic knowledge. In: TARK, pp. 177–186 (2011)
27. Huang, X., van der Meyden, R.: Symbolic model checking epistemic strategy logic. In: AAI, pp. 1426–1432 (2014)
28. Jansen, D.N.: Understanding Fox and Glynn’s “Computing Poisson probabilities”. Technical report. ICIS-R11001, Institute for Computing and Information Sciences, Radboud Universiteit (2011)
29. Kacprzak, M., et al.: Verics 2007 - a model checker for knowledge and real-time. *Fundam. Informaticae* **85**(1–4), 313–328 (2008)
30. Katoen, J., Khattri, M., Zapreev, I.S.: A Markov reward model checker. In: QEST, pp. 243–244 (2005)
31. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006). [https://doi.org/10.1007/11871842\\_29](https://doi.org/10.1007/11871842_29)
32. Kong, J., Lomuscio, A.: Model checking multi-agent systems against LDLK specifications. In: IJCAI, pp. 1138–1144 (2017)
33. Kong, J., Lomuscio, A.: Model checking multi-agent systems against LDLK specifications on finite traces. In: AAMAS, pp. 166–174 (2018)
34. Křetínský, J., Meggendorfer, T., Sickert, S., Ziegler, C.: Rabinizer 4: from LTL to your favourite deterministic automaton. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 567–577. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-96145-3\\_30](https://doi.org/10.1007/978-3-319-96145-3_30)
35. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22110-1\\_47](https://doi.org/10.1007/978-3-642-22110-1_47)
36. Lomuscio, A., Paquet, H.: Verification of multi-agent systems via SDD-based model checking. In: AAMAS, pp. 1713–1714 (2015)

37. Lomuscio, A., Pecheur, C., Raimondi, F.: Automatic verification of knowledge and time with NuSMV. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, 6–12 January 2007, pp. 1384–1389 (2007)
38. Lomuscio, A., Pirovano, E.: Verifying emergence of bounded time properties in probabilistic swarm systems. In: IJCAI, pp. 403–409 (2018)
39. Lomuscio, A., Pirovano, E.: A counter abstraction technique for the verification of probabilistic swarm systems. In: AAMAS, pp. 161–169 (2019)
40. Lomuscio, A., Pirovano, E.: Parameterised verification of strategic properties in probabilistic multi-agent systems. In: AAMAS, pp. 762–770 (2020)
41. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: a model checker for the verification of multi-agent systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02658-4\\_55](https://doi.org/10.1007/978-3-642-02658-4_55)
42. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. *Int. J. Softw. Tools Technol. Transfer* **19**(1), 9–30 (2015). <https://doi.org/10.1007/s10009-015-0378-x>
43. Lomuscio, A., Raimondi, F.: MCMAS: a model checker for multi-agent systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 450–454. Springer, Heidelberg (2006). [https://doi.org/10.1007/11691372\\_31](https://doi.org/10.1007/11691372_31)
44. Lv, G., Su, K., Xu, Y.: CacBDD: a BDD package with dynamic cache management. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 229–234. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-39799-8\\_15](https://doi.org/10.1007/978-3-642-39799-8_15)
45. de Moura, L., Björner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
46. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. *Logical Methods Comput. Sci.* **3**(3) (2007)
47. PRISM web site. <http://www.prismmodelchecker.org>
48. Schewe, S.: Tighter bounds for the determinisation of Büchi automata. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 167–181. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-00596-1\\_13](https://doi.org/10.1007/978-3-642-00596-1_13)
49. Schewe, S., Varghese, T.: Tight bounds for the determinisation and complementation of generalised Büchi automata. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 42–56. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33386-6\\_5](https://doi.org/10.1007/978-3-642-33386-6_5)
50. Sickert, S., Křetínský, J.: MoChiBA: probabilistic LTL model checking using limit-deterministic Büchi automata. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 130–137. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-46520-3\\_9](https://doi.org/10.1007/978-3-319-46520-3_9)
51. Somenzi, F.: CUDD: CU decision diagram package release 2.5.0. <http://vlsi.colorado.edu/~fabio/CUDD/>
52. Su, K., Sattar, A., Luo, X.: Model checking temporal logics of knowledge via OBDDs. *Comput. J.* **50**(4), 403–420 (2007)
53. Tarjan, R.E.: Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1**(2), 146–160 (1972)
54. Vahidi, A.: JDD, a pure Java BDD and Z-BDD library. <http://javaddlib.sourceforge.net/jdd/>